Software Engineering Institute

Software Assurance Measurement—State of the Practice

Dan Shoemaker, University of Detroit Mercy
Nancy R. Mead, Software Engineering Institute

November 2013

TECHNICAL NOTE
CMU/SEI-2013-TN-019

CERT Division

http://www.sei.cmu.edu

Carnegie Mellon University

This material is based upon work funded and supported by Department of Homeland Security under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Department of Homeland Security or the United States Department of Defense.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

DM-0000745

Table of Contents

Acknowledgments

The authors thank the following individuals for their contributions to this report. We greatly appreciate their insights and efforts.

We thank our sponsor, Joe Jarzombek, U.S. Department of Homeland Security (DHS Office of Cybersecurity and Communications), who had the insight to recognize the need for this report and support its initial development.

The following individuals provided critical insights in their review of this document:

- Glenn Johnson, (ISC)^2
- Andrew Kornecki, Embry-Riddle Aeronautical University

In addition, we thank Hollen Barmer of the Software Engineering Institute for her editorial support.

Abstract

This report identifies and describes the current state of the practice in software assurance measurement. This discussion focuses on the methods and technologies that are applicable in the domain of existing software products, software services, and software processes. This report is not meant to be prescriptive; instead it attempts to provide an end-to-end discussion of the state of the practice in software assurance measurement. In addition, it points out significant emerging trends in the field. The overall discussion touches on the existing principles, concepts, methods, tools, techniques, and best practices for detection of defects and vulnerabilities in code.

1 Background and Assumptions

1.1 The Importance and Challenges of Software Assurance Measurement

An earlier report notes that “commonly used software engineering practices permit dangerous defects that let attackers compromise millions of computers every year” [PITAC 2005]. This appraisal raises a practical question for the industry: “Given the critical importance of software, why aren’t we doing a better job of securing it?” The answer lies in the nature of software itself.

Software programs are both invisible and extremely intricate. More importantly, the act of writing computer code is just that—“writing.” Because the product is so complex and programming is a creative act, it is almost impossible to ensure product consistency. As a result, although we have produced trillions of lines of code over the past 55 years, we really don’t know what is happening inside that vast landscape. Likewise, it should go without saying that the security of our software products will not improve until we can get better command of their production. Enhanced control requires improved oversight, which is the reason why assurance measurement is such an important topic.

1.2 Why Security Is a Concern

The lack of visibility into commercial code raises serious concerns for our way of life, since the one thing we have learned for certain over the past half century is that unknown or undocumented code will contain defects [Humphrey 1994]. The problem is that, due to the size of most businesses’ software portfolios, finding the few exploitable defects in all of the clean code is a very difficult thing to do. That is the reason why it has been impossible to say with certainty that a given application is secure. In fact, it is much easier to simply assume that any application, no matter how rigorously developed, will have some exploitable flaw that could cause a security problem.

The basis for this assumption lies in the fact that software development is not a manufacturing process. If computer applications were built like automobiles, there would be obvious inspection points, and every inspector would know exactly what to look for at any point in the process. Thus, we can say with assurance that a given automobile is defect-free because our inspectors have confirmed that with the naked eye. However, the correctness of a line of computer code depends on the innate abilities of the programmer who wrote it, and those abilities are difficult to standardize or replicate. Testing might confirm that a given program functions correctly, but without carefully analyzing every line of code in that program, we cannot say with absolute certainty that it does not contain harmful defects or intentionally inserted malicious lines of code.

We do not take the effort to fully ensure software products because of two cultural factors: consumer expectations about price and business requirements for profit. Both of these factors are rooted in the reality that the extra time it takes to do assurance properly is overhead to the production process, which increases the cost of creating the product. Because customers are used to inexpensive software products, it is likely that when given the choice between “cheaper” and “less risky,” most people would choose “cheaper.” Similarly, if the producer doesn’t pass the cost of higher assurance along to the consumer, the overhead cuts into the producer’s profit margin, and reduced profitability is a scenario most companies are unwilling to accept. Thus, considering these factors, a dollars-and-cents case must be made for the added cost of software assurance.

The need to balance cost with benefit is the reason why some form of standard, systematic, measurement-based data gathering process is so important to the overall effort to ensure code. Accurate and reliable software assurance data would allow organizations to objectively judge the effectiveness of their software development process and the correctness of their products. Also, that standardized data could provide lessons learned that could make the overall development and sustainment of software more effective and efficient.

Nevertheless, the intangibility of software makes it difficult to measure. With tangible objects, like an automobile, we have well-known—and in many cases ancient—measures. For example, we can answer a question like “How powerful is it?” with “200 horsepower V8.” Answers to such questions provide all of the information required to make practical business decisions about the automobile’s power plant. However, for something as abstract as software, rigorous research is needed to define *what* to measure before determining how to measure it and how that measurement translates to meaningful information about the product.

Persistent and commonly understood measures help make the software life cycle more visible. Previously, the only way to describe the life cycle of a software product was in management terms, such as “a $5 million development,” or “a two-year project.” These measures are useful to conventional business people because that information helps them make decisions about the product during its development and use. On the other hand, without standard, meaningful measures, software engineers are stuck with guesswork when it comes to ensuring product correctness. Standard measures define concrete, quantifiable attributes. Reliable data about those how those attributes are functioning allows software engineers to understand what is really happening in the development and sustainment work. With that increased understanding the potential exists for better assurance throughout the life cycle.

Measurement is the ability to objectively observe and quantify the outcomes of a given product or process. Those outcomes may represent any number of abstract variables, such as defects, lines of code, or even time. The measurement challenge is to turn an existing set of outcomes into standard and commonly accepted measures that can be used to accurately gauge the security of the product and its processes. Measurement provides the link between the pragmatic, real-world development and sustainment processes and the requirement that we must understand what is happening inside the process in order to guarantee the security of the product. The right set of standard measures ensures that engineers are able to monitor the right things during the process in order to make those assurances. In that respect, the ways to obtain consistently accurate data have become a critical part of the overall discussion about how to assure code.

The question still remains: How do you describe software in ways that are meaningful to engineers and developers, as well as managers? There are also practical considerations: What data should we collect? What are the appropriate units of measurement, and what are the measurement intervals? All of these questions must be answered to create and execute an effective software assurance process.

1.3 Purpose of This Report

The purpose of this report is to identify and describe the current state of the practice in software assurance measurement. This discussion focuses on the methods and technologies that are applicable within the domain of existing software products, software services, and software processes. This report is not meant to be prescriptive; instead it attempts to provide an end-to-end discussion of the state of the practice in software security measurement. In addition, it points out significant emerging trends in the field. The overall discussion touches on the fundamental principles, concepts, methods, tools, techniques, and best practices for detection of defects and vulnerabilities in code. It does not specifically endorse products or present research findings.

Although the concept of software assurance measurement is still in its infancy, there are 30 years of accumulated best practices that apply specifically to defect identification and removal. Because the primary goal of assurance is to ensure that exploitable defects are not present in code, most of these practices are relevant to this discussion. Moreover, many of the strategies, methods, tools, and techniques from the domain of quality assurance also apply to defect identification and measurement in the security domain. Therefore, we feel that it is justifiable to use the extensive literature of software quality assurance to support the creation of a body of knowledge in software security assurance.

Where the practices related to quality assurance differ from those related to security assurance is in the existence of the adversary. With quality, the concern is correctness; with security, the presence of an adversary creates additional problems such as malware and malicious additions to code. The process for assessing the security of code must account for these additional factors. Therefore, we stress the importance of distinguishing between correctness and exploitation throughout this report.

1.4 Intended Audience

The intended primary and secondary audiences for this document are best described in terms of professional roles [NICCS 2013]. The primary audience includes

- chief information officers (CIOs)
- chief information security officers (CISOs)
- software project managers
- software engineers
- software testers and maintainers
- software technology demonstrators
- personnel who are performing software services

Readers in the following roles are the intended secondary audiences:

- system engineers
- system and code integrators
- information assurance professionals
- legal and regulatory compliance professionals
- policy makers

- risk managers
- accreditors and certifiers
- information security auditors and evaluators
- professionals involved in training and educating assurance personnel

The general aim in addressing these audiences is to help them understand how to use standard measurement data to characterize the security properties of software products and system components. That quantitative understanding will help them better assess the impact of threats and vulnerabilities on the security of their enterprises. More specifically, these audiences should be able to understand and gauge how defective software can be subverted or sabotaged to cause physical harm and financial loss to their businesses.

1.5 Scope

This report focuses on the body of knowledge in software measurement, which has been derived from a very broad range of research studies that began in the 1960s with rudimentary software quality assurance practices. Any currently appropriate methods, tools, techniques, and initiatives that have been demonstrated to produce defect-free and properly designed software are considered to apply here, regardless of their original purpose. Also covered in this discussion are the technologies, tools, and other mechanisms that have been used by pertinent organizations in the measurement and management of software security. Finally, assessment and measurement methods that have been used to ensure the security and integrity of sourced products and services—also known as supply chain risk management—are within the scope of this report.

Any method, tool, or technique that is focused on improving software quality, reliability, or safety is considered in scope as long as it is used for the express purpose of generating and collecting measurement data. The methods and models, techniques and tools, and automated means that have traditionally been associated with the field of information assurance are considered out of the scope of this report; examples of out-of-scope items include authentication, authorization, access control, and network security in software-intensive systems. Standard-based certification models such as Federal Information Processing Standards (FIPS) 200 and the Common Criteria (CC) evaluations are also out of the scope of this document.

In summary, this document discusses only those methodologies, techniques, and tools that have been designed for—or adapted/reapplied to—the measurement of the internal or external characteristics of software products and/or processes. The discussions in this report include a brief overview of measurement as it was originally conceived and a presentation of the current or proposed ways measurement can be used to support security assurance. The overview is intended to provide context and comprehensive knowledge of all concepts and technologies used in the assessment of the security of code. Various lists of those concepts and technologies are provided in the extensive set of appendices included with this document. The purpose of these appendices is to offer a comprehensive range of summaries of useful methods and tools for software assurance measurement.

1.6 Constraints

The state of the practice that is reported here reflects the time frame in which the source information was collected. This report discusses the activities, practices, technologies, tools, and initiatives that apply to professionals who develop software. This includes requirements analysts, architects, designers, programmers, and all types of testing professionals. The report also addresses sustainment and acquisition issues relevant to assurance and assurance issues as they apply to supply chain risk management.

Excluded is consideration of specialized business environments in which software might be created or operated, such as classified projects. The report does not address operational security, access control, or personnel security concerns, and it does not address non-technical, business-oriented risk management considerations except to the extent that those risks might affect the assurance of the software sustainment processes of the organization.

Finally, the practices, tools, and knowledge required to satisfy a range of clearance levels are presumed to differ mainly in the rigor of their application. The report may acknowledge those differences in instances where that information is relevant. However, the report does not delve into unique approaches, tools, or other considerations for particular types of security situations.

2 Definition of Terms

2.1 What Is Software Assurance?

There are many definitions of the term *software assurance*. In this report, we use the following definition: “the level of confidence that software is free from vulnerabilities either intentionally designed into the software or accidentally inserted at any time during its life cycle and that the software functions as intended” [CNSS 2010].

Definitions provided by other organizations mirror this definition in concept. The following three additional concepts can be considered: “planned and systematic,” “software process [assessment],” and “conformance with requirements, standards, and procedures” [NIST 2013, NASA 2013]. The Object Management Group (OMG) System Assurance Task Force elaborates by defining the following terms [OMG 2013]:

- *vulnerability*—an implementation, design, or requirements flaw that an attacker has access to and is capable of exploiting
- *threat*—any potential hazard or harm to the data, systems, or environment that can occur by leveraging a vulnerability
- *risk*—the probability of the threats using the vulnerabilities
- *exposure*—the damage done when a threat takes advantage of a vulnerability

In addition, the OMG makes it clear that software assurance is not just about correctness. “The view of correctness has to be supplemented by the identification of *discrete critical programming errors*” [OMG 2013]. These errors are the result of bad practice, which makes a system unsuitable for use regardless of its apparent correctness. In essence, these vulnerabilities might not fail a test for correctness, but under specific circumstances they can lead to catastrophic outages, performance degradations, security breaches, corrupted data, and myriad other problems [OMG 2013]. The cause of critical programming errors is some combination of failure to comply with good architectural design principles and failure to employ good coding practice. These failures are normally detected by measuring the static attributes of an application [OMG 2013].

2.2 What Is Software Assurance Measurement?

Software assurance measurement assesses the extent to which a system or software item possesses desirable characteristics [NDIA 1999]. The goal of software assurance measurement is to “identify and analyze anomalies and defects in software that can cause failure, and to describe appropriate responses to threats so as to minimize the risk of failure” [NDIA 1999]. To accomplish this task, software assurance measurement collects and analyzes quantitative indicators against which trust can be developed and subsequently validated [NIST 2013]. Therefore, the purpose of software assurance measurement is to “establish a basis for gaining justifiable confidence (trust, if you will) that software will consistently demonstrate one or more desirable properties” [IATAC 2009]. This basis includes such properties as quality, reliability, correctness, dependability, usability, interoperability, safety, fault tolerance, and security [IATAC 2009].

Software assurance measurement is based on metrics. A metric is “a standard of measurement, a mathematical function that associates real nonnegative numbers” [Merriam-Webster 2013]. The general aim of metrics is to describe a specific given quality of an object [SQA 2013]. The practical aim of a metric is to evaluate the performance of an object in a real-world setting [IATAC 2009]. The specific purpose of software metrics is to evaluate the performance of a specified life-cycle activity or function [OMG 2013]. The evaluation can be qualitative or quantitative, or a mix of both [OMG 2013]. Software metrics apply to both the process and the product characteristics within the life cycle. However, in all instances, the measurable attributes of the object must be capable of being confirmed [OMG 2013].

2.3 Measurement Domains

Software measurement relies on discrete indicators to support real-world decision making. A software assurance indicator is a metric or combination of metrics that provides useful information about the development process, the conduct of the project, or the characteristics of the product itself [OMG 2013]. Software measurement takes place in two domains: internal and external: “The characteristics that are evaluated by external assessments are those that face the user community. Characteristics evaluated by internal assessments do not” [SQA 2013].

External assessments tend to be behavioral in nature because they are centered on interaction. They measure characteristics that can be described only in terms of how an object relates to its environment [SQA 2013]. Internal assessments typically involve direct measures, which describe characteristics that can be observed. In essence, internal metrics are gathered by investigating how the product behaves, regardless of its environment [SQA 2013].

One of the major benefits of internal product metrics is that they tend to be quantitative, and therefore empirical data can be collected for them [SQA 2013]. Quantitative data can also be collected by automated tools [SQA 2013]. Finally, because internal assessments tend to produce empirical data, the results can be analyzed using statistical methods.

External assessments allow decision makers to evaluate functional characteristics of the product and process, such as reliability, efficiency, and security. However, external assessments can also involve the static analysis of the structure of the product. For instance, reliability in software denotes the level of risk and the likelihood of potential application failure. Reliability is a non-tangible attribute of resilience and structural solidity, so it cannot be directly observed [IEEE 2012]. Therefore, depending on the application architecture and the third-party components, external assessments of reliability are most often aimed at confirming compliance with specified requirements, design criteria, and coding best practices [IEEE 2012]. Efficiency, on the other hand, can be assessed behaviorally and also through direct observation. Efficiency is most often characterized by behavioral factors such as some measure of performance over time. However, factors such as processing bottlenecks, future scalability problems, source code concerns, and software architectural design issues can be directly evaluated through static analysis. Both behavioral and internal performance efficiency evaluations typically involve examining at least the following software engineering best practice and technical attributes [CNSS 2010]:

- application architecture practices
- appropriate interactions with expensive and/or remote resources
- data access performance and data management

- memory, network, and disk space management
- coding practices
- compliance with object-oriented and structured programming best practices
- compliance with Structured Query Language (SQL) programming best practices
- ensuring centralization of client requests to reduce network traffic
- avoiding SQL queries that do not use an index against large tables in a loop

Most security vulnerabilities, such as SQL injection or cross-site scripting, result from poor coding and/or design [CNSS 2010]. Those factors can be observed through static analysis. Therefore, security evaluations focus on the description of the status of known vulnerabilities. Typically, evaluations that are aimed at determining system security characterize the risks associated with improper coding and incorrect architectural design practices. Because they can be observed and described, common types of security vulnerabilities are also documented and maintained in lists [MITRE 2013a, SEI 2013]. In general, security measurement involves assessment of the following software characteristics, software engineering practices, and technical attributes [Woodley 2013, Fenton 1998]:

- application architecture practices
- multi-layer design compliance
- security best practices (e.g., input validation, SQL injection, cross-site scripting)
- programming practices (code-level)
- error and exception handling
- security best practices (e.g., system functions access, access control to programs)
- size
- avoiding fields in servlet classes that are not final static
- avoiding data access without including error management
- checking control return codes and implementing error handling mechanisms
- ensuring input validation to avoid cross-site scripting flaws or SQL injections

2.4 Critical Programming Errors

Software assurance always involves the elimination of critical programming errors. But the assurance process does not concentrate exclusively on code. Comprehensive and accurate identification and mitigation of critical programming errors requires the assessment of the complete set of structural attributes involving the application's architecture, code, and documentation [CNSS 2010]. Each of these characteristics affects, and is in turn affected by, application attributes at numerous levels of abstraction. Places where critical programming errors can occur include the source code, architecture, and program framework, as well as the database schema, which defines the conceptual and logical architecture of a system.

The analysis that seeks to identify and eliminate critical programming errors is different from local, component-level code analysis. That is because local, code-level analysis focuses primarily on code implementation concerns such as bug finding and performance testing [Wikipedia 2013]. The analysis aimed at identifying and mitigating critical programming errors focuses on application-level attributes, which include the following software engineering best practices and technical features [Wikipedia 2013]:

- application architecture practices
- coding practices
- complexity of algorithms
- complexity of programming practices
- compliance with object-oriented and structured programming best practices
- dirty programming
- error and exception handling (for all layers—graphical user interface [GUI], logic, and data)
- multi-layer design compliance
- resource bounds management
- software's avoidance of patterns that lead to unexpected behaviors
- software's management of data integrity and consistency
- transaction complexity level
- component or pattern reuse ratio

3 Measurement and Metrics

3.1 Definitions of Terms

Measurement captures concrete values that characterize some attribute of the software. Measurement uses metric data as input and produces metric information as output. The process of applying a metric in the assessment of a product or a process results in a data item. The generation of consistent metric data over the course of the development process results in a data set.

There are three kinds of metrics that are a part of software measurement: product metrics, process metrics, and resource metrics. Product metrics are directly associated with the product itself. Product metrics attempt to measure product security or characteristics of the product that can be connected with product trustworthiness. Process metrics concentrate on the process and measure non-tangible attributes such as rigor, correctness, and reliability. The aim of process metrics is to either distinguish problems in the execution of the process or define effective practices. Finally, resource metrics describe the pragmatic business elements required for the development and implementation of software systems, such as time, effort, and cost.

Ideally, a metric should possess five essential properties [Wikipedia 2013]:

- *simple*—The definition and use of the metric is uncomplicated.
- *objective*—Different people arrive at the same value when executing the measurement process.
- *easily collected*—The cost and effort of data collection is reasonable and feasible.
- *robust*—The values produced by the metric are insensitive to irrelevant changes in conditions.
- *valid*—The metric accurately characterizes the target it is supposed to measure.

There are two generic classes of metrics that can be useful to assurance. The first class is management metrics, which provide data about the process itself, including any relevant performance or productivity issues. The second class of metrics defines some aspect of the current status of the product or process. Metrics that describe the status of the product involve data about the specific condition of a software artifact including integrity (absence or presence of defects), architecture (design), and the satisfaction of process and resource requirements.

Management metrics monitor and control all aspects of the life cycle. They are used to assess process performance and qualitative factors like product security and quality; they are also used to assess resources, cost, and task completion. On the other hand, status metrics are used to describe or estimate product characteristics, product performance, and product quality concerns.

There are many types of potential metrics for both of these general areas. As a result, users must define and employ measurement criteria to decide which metric best fits the needs of the project. For instance, some metrics such as “defects” can serve both management and status measurement purposes. Thus, a distinction must be made about what a metric like “defect” means in terms of the execution of the process (management), or the status of the product. Criteria to make such distinctions are defined during a formal metric development process.

3.2 The Metric Development Process

Metric development begins with the determination of the appropriate measure to use to characterize a given attribute within a particular measurement situation. An attribute is a measurable characteristic of a target for assessment. The goal of the metric development process is to select measures that have consistent and unambiguous meanings so that the implications of a particular measurement can be precisely understood. For example, the metric “lines of code” does not provide sufficient information to ensure consistent understanding among different interpreters of the data. Questions would include “How do you define a line of code?” and “Which lines of code are to be included in the count—source, object, assembly, comment, etc.?” Therefore, the metric development process must provide a clear and usable understanding of the meaning of all values that are used in the measurement process.

In order to be useful, software assurance metric data must be consistently valid, retrievable, relevant, and cost effective. In order to be practically effective, the data must be retrievable, understandable, and relevant. Data must provide concrete values. Consequently, data collection must be based on consistent units of measure that are capable of being normalized. Moreover, once data has been collected, it must be stored so that it retains its usefulness. The generic requirements for metrics-based assurance are the following:

- agreed-on metrics to define the assurance
- scientifically derived data for each metric
- sufficient justification for collecting data to account for the added costs of collection

A frame of reference must be defined and adopted before assessors can select the particular metrics that will be used for the data collection. That frame of reference provides the practical justification for the metric selection process. A frame of reference is nothing more than the logic (or justification) for choosing the metrics that are eventually employed. For instance, there are no commonly agreed on measures to assess security; however, a wide range of standard metrics, such as defect counts or cyclomatic complexity, could be used to measure that characteristic. The metric that is selected depends on the context of its intended use.

The only rule that guides the actual selection and adoption process is that the metrics must be objectively measurable, produce meaningful data, and fit the frame of reference that has been adopted. For instance, security is often characterized in terms of exploitable defects. That characterization is objectively measurable and therefore produces data (in terms of counts), that can provide a clear understanding of the situation it is employed to describe (e.g., potential vulnerabilities).

3.3 Functional and Structural Attributes of Correctness

Software measurement assesses two related but distinctly different attributes: functional correctness and structural correctness. Functional correctness measures how the software performs in its environment. Structural correctness assesses the actual product and process implementation.

Software functional correctness describes how closely the behavior of the software complies with or conforms to its functional design requirements, or specifications. In effect, functional correctness characterizes how completely the piece of software achieves its contractual purpose.

Functional correctness is typically implemented, enforced, and measured through software testing. The testing for correctness is done by evaluating the existing behavior of the software against a logical point of comparison. In essence, the logical point of comparison is the basis that a particular decision maker adopts to form a conclusion. Logical points of comparison include such diverse things as “specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria” [IEEE 2000].

Measurements of structural correctness assess how well the software satisfies environmental or business requirements, which support the actual delivery of the functional requirements. For instance, structural correctness characterizes qualities such as the robustness or maintainability of the software, or whether the software was produced properly. Structural correctness is evaluated through the analysis of the software’s fundamental infrastructure, and the validation of its code against defined acceptability requirements at the unit, integration, and system level. Structural measurement also assesses how well the architecture adheres to sound principles of software architectural design [OMG 2013, “Software Environment”].

3.4 Methods and Models

As technology has matured over the past decade, root causes of insecure and unreliable software have been identified earlier in the life cycle. This early identification has been due to improvement in the use of formal measurement methods. Formal measurement methods are built around a set of variables that are capable of capturing and describing discrete characteristics of interest at early stages in the life cycle. Examples of such early-stage variables would be requirements factors like correct elicitation and abstraction of requirements, and design factors like cohesive and coherent modules and properly implemented information hiding.

Those variables can be chosen and combined into a model of assurance for a given evaluation target. In practical terms, this means that the general security of a piece of software can be evaluated by constructing a measurement model, which embodies a set of metrics that target explicit matters of concern in the product and process. That model itself is a conceptual, usually quantitative, array of variables that accurately and appropriately represent the relationship between the chosen attributes of a given target for assessment—for instance, the cost to develop a piece of software versus the level of assurance it implements.

There are three major categories of measurement models: descriptive, analytic, and predictive. Measurement models of any type can be created by equations, or analysis of sets of variables, which characterize practical concerns about the software. A good measurement model allows users to fully understand the influence of all factors that affect the outcome of a product or process, not just primary influences. A good measurement model also has predictive capabilities; that is, given current known values, it predicts future values of those attributes with an acceptable degree of certainty.

Assurance models measure and predict the level of assurance of a given product or process. In that respect, assurance models characterize the state of assurance for any given piece of software. They provide a reasonable answer to questions such as “How much security is good enough?” and “How do you determine when you are secure enough?” Finally, assurance measurement models can validate the correctness of the software assurance process itself. Individual assurance models are built to evaluate everything from error detection efficiency, to internal program failures, software reliability estimates, and degree of availability testing required. Assurance models can also be used to assess the effectiveness and efficiency of the software management, processes, and infrastructure of the organization.

Assurance models rely on maximum likelihood estimates, numerical methods, and confidence intervals to make their assumptions. Therefore, it is essential to validate the correctness of the model itself. Validation involves applying the model to a set of historical data and comparing the model's predicted outcomes to the actual results. The output from a model should be a metric, and that output should be usable as input to another model.

Assurance modeling provides a quantitative estimate of the level of trust that can be assumed for the system. Models that predict the availability and reliability of the system include estimates such as initial error counts and error models incorporating error generation, up-time, and the time to close software error reports.

Software error detection models characterize the state of debugging of the system, which encompasses such concerns as the probable number of software errors that will be corrected at a given time in system operation, plus methods for developing programs with low error content and for developing measures of program complexity.

Models of internal program structure include metrics such as the number of paths (modules) traversed, the number of times a path has been traversed, and the probability of failure, as well as advice about any automated testing that might be required to execute every program path.

Testing effectiveness models and techniques provide an estimate of the number of tests necessary to execute all program paths and statistical test models. Software management and organizational structure models contain statistical measures for process performance. These models can mathematically relate error probability to the program testing process and the economics of debugging due to error growth.

Because assurance is normally judged against failure, the use of a measurement model for software assurance requires a generic and comprehensive definition of what constitutes "failure" in a particular measurement setting. That definition is necessary in order to incorporate considerations of every type of failure at every severity level. Unfortunately, up to this point, a large part of software assurance research has been devoted to defect identification, though defects are not the only type of failure. This narrow definition is due partly to the fact that defect data is more readily available than other types of data, but mostly because there have not been popular alternatives for modeling the causes and consequences of failure for a software item. With the advent of exploits against software products that both run correctly and fulfill a given purpose, the definition of failure must be expanded. That expansion would include incorporating the following into the definition of failure: intentional (backdoors) or unintentional (defects) vulnerabilities that can be exploited by an adversary, as well as the mere presence of malicious objects in the code.

3.5 Measurement Management

Measurement management ensures that suitable measures are created and applied as needed in the life cycle of the software or system. Measurement management requires the formulation of coherent baselines of measures, which accurately reflect product characteristics and life-cycle management needs. Measurement baselines must be capable of being revised as needed to track the evolving product throughout its life cycle. Using the appropriate set of baselines, the measurement management process ensures that the right set of metrics is always in place to produce the desired measurement outcomes.

The measurement management process then collects data about discrete aspects of the product and/or process functioning to support operational and strategic decision making about the product. The measurement management process should provide data-driven feedback in real-time or near-real-time to project and development managers and to the technical staff. In addition, measurement management should support the decision making of organizational-level managers. Finally, the measurement management process should also ensure buy-in from all appropriate stakeholders throughout the life cycle.

The following general goals and/or feasibility constraints are important to ensuring the successful creation of measurement baselines and the use of measurement data to support operational and management decision making:

- effective application of product and project measures throughout the life cycle
- optimization of the development of assured software within constraints
- maximizing software and system assurance in its actual use environment

Practically speaking, it is necessary to manage the measurement of evolving product, process, cost, and schedule issues in the same way that every other organizational function is tracked and managed. That includes the development of appropriate measures within constraints: fitting proposed measures to available resources and time as well as staff capability. Finally, any relevant contract and application criticality constraints must be considered in the development and use of measures. In addition to development constraints, software engineering process constraints must also be accounted for (for example, the prevalence of subcontractors and the amount of reuse or legacy work).

3.6 Managing Through Measurement Baselines

Management control is typically enforced through measurement baselines. A measurement baseline is a collection of discrete metrics that characterize an item of interest for a target of measurement. Using a measurement baseline, it is possible to make meaningful comparisons of product and process performance in order to support management and operational decision making. Such comparisons can be vital to improving products and processes over time through the use of predictive and stochastic assurance models for decision making. Potential metrics in support of such an effort might include basic measures such as defect data, productivity data, and threat/vulnerability data. An evolving baseline model would provide managers with the necessary assurance insight, cost controls, and business information for any process or product, and it would allow value tracking for the assurance process, which is a major strategic advantage for managers.

There are four steps involved in formulating a measurement baseline. The first step is for the organization to identify and define the target of measurement. The organization then establishes the requisite metrics to assure the desired measurement objectives and places those measures in a formally controlled baseline. Then, based on the aims of the metrics, the organization establishes the comparative criteria for tracking performance. Those criteria establish what will be learned from analyzing the data from each metric over time. Finally, the organization carries out the routine measurement data collection and analysis activities, using the baseline metrics.

The establishment of the target for measurement is a straightforward process. Managers decompose the product or process that is to be baselined and tracked. The manager then assigns meaningful measures to characterize items of interest for each component. The characterization itself normally supports business goals. Next, the manager arranges the measures into a logical array of items for comparative and/or descriptive measurement purposes and develops objective criteria for assessing the status of each measure. The manager then formalizes the measurement baseline as a standard cost and budgeting function for the organization as a whole. The outcome of this process is a credible, metric based, assurance operation that allows the manager to make rational judgments about any risks involved with the target of measurement.

The next step in the process is to ensure the consistent and rigorous collection of operational data for each metric. That also includes ensuring that all changes in the definition and/or collection of the discrete data items that make up the baseline are controlled. The aim is to assure reliability and integrity in the collection process. Things to consider include assurance that the data that is collected from the performance of the day-to-day work remains valid and continues to appropriately characterize risk. Finally, a means must be specified to ensure that the data that is gathered continues to be responsive to and capture technical and programmatic performance needs.

Once the consistent collection of data can be assured, a standard basis for evaluating performance must be defined for each metric. This is necessary to ensure meaningful reporting of results for measures of interest over time. It is normal for stakeholders to define the actual criteria for evaluation at this stage. Nevertheless, those criteria must be revisited over time in order to confirm their continuing validity.

The performance measurement process itself must be performed as a standard organizational operation throughout the life cycle. Establishing the routine elements of the operation involves the preparation of everyday and custom reports for decision makers. Because maintenance of the integrity of the baseline is the critical requirement for the success of the overall process, rigorous control over the measurement baseline should be emphasized.

Measurement baseline management ensures the continuing integrity of the representation of the target for measurement. Measurement baselines contain formal and discrete metric items that uniquely characterize the target for measurement. In order to be effective, the management of the measurement baseline must be conducted in a standard and uniform fashion during all phases of the life cycle. This means that once established, the integrity of the measurement baseline must be maintained through the life cycle. Measurement baseline management is applicable to all types of measurement situations where formal metrics are used. For the sake of assessment, it is common for a series of measurement baselines of increasing complexity to be established as the target for measurement evolves over time.

The measurement baseline identification scheme is the cornerstone of measurement management. This scheme is usually finalized during design of the overall measurement program. All discrete and measurable components of the measurement target are identified. As metrics are assigned to those components, those metric items are uniquely labeled and then arrayed in a unified and coherent baseline of relevant measures that characterize the target for measurement. That array is based on an appropriate set of interrelationships and dependencies. The array then represents the basic structure, or configuration, of the measures for that particular target of measurement.

By definition, the metrics in each baseline comprise any measures that produce data and which have been explicitly placed under baseline control. In practice the item is created by giving it a formal, unique descriptor label. Then, baseline control ensures the integrity of each metric added to the baseline. Although the decisions that determine the actual required information are made outside of the metric baseline management process, the creation and control of a metric baseline is an absolute prerequisite for the successful use of measurement to support decision making.

The most common organizational model for a metric baseline is a hierarchy of aggregate measures. Management personnel are assigned to the control of one or more levels in this hierarchy. There can be parallel hierarchies of product metrics (e.g., a system, program, or package), component metrics (e.g., a subsystem, unit, or program function), and unit metrics (e.g., a procedure, routine, or module) under control at one time for a target of measurement.

Since the same metric can exist at multiple levels in the hierarchy, each individual metric item must be given a unique and appropriate identification label. Label numbering is generally associated with the structure itself. Thus, the labeling can be used to designate and relate the position of any given metric to the overall “family tree” of the product. At a minimum, a label should be composed of an aggregate of the following designations (fields): unique item designation, baseline designation, and product/version designation.

Structural change takes place when a new metric is added or when the meaning of an existing metric is changed. Over time, the aggregate of metrics that comprise an evolving structure is reviewed or audited to verify logical consistency. Where required and appropriate, any new item that is added to the aggregate gets a new identification label to reflect its new baseline. It is acceptable to have a number of metric baselines under control for the same product. However, the management level authorized to approve change to each metric baseline must be explicitly defined. Authorization is always given at the highest practical level in the organization. Also, changes at any level in the basic structure must be appropriately maintained at all affected levels.

The metric baseline change control process is composed of change accounting and measurement library management. Metric baseline change management is made up of change authorization, verification control, and release processing. Metric baseline configuration verification encompasses procedures for status accounting and the verification of compliance with change authorizations. Requests for change must be initiated on an organizationally standard request form. This form is an official document that is developed as part of the overall metric baseline creation process. The authorizations for any change to a baseline item are granted by means of a change authorization form. The change authorization form is also a formal document that is developed at the same time the change request form is created. All requests for changes or additions to baselines are given a unique number. That number must correspond to the change authorization.

3.7 Measurement Data Acquisition, Retention, and Use

The business purpose for capturing and analyzing software measurement data is to support management decision making with respect to the trustworthiness of the product and process. In effect, objective data helps managers to better monitor the integrity and correctness of the software process, as well as ensure that the actual work does not produce exploitable defects. At the operational level data allows project managers to more accurately estimate cost, staff effort, and schedule commitments.

It is particularly essential to validate all data that is used for assurance measurement because the decisions made using measurement data have real-world impacts. Thus, the data must be regularly evaluated to ensure its accuracy and appropriateness. In addition, the logical structure and correct association of data items in the baseline must also be routinely demonstrated.

Though regular validation is required for software assurance measurement, a number of practical problems affect the validation of data. First, the process of data collection and validation adds to the cost of a development project. As a result, dedicated data validation often must be justified to management. Second, software measurement data is often considered proprietary, especially when cost and productivity information can be derived from the data. Therefore, it is difficult to get authorization to regularly access the data. Third, companies do not want to release assurance information that their competitors might see or use. In addition, companies do not want to create a potential conflict between the data they collect and official reports of a project’s progress or status. Thus, many software development organizations now have in-house databases of project experience data that they control but will not validate.

4 Standard Assurance Measurement Methodologies

4.1 Measurement Processes

Security measurement should employ a commonly accepted and standardized process to conduct baselining, analysis, and validation. To be optimally useful, any measurement program should span the entire life cycle. In addition, the generic measurement process itself should produce meaningful data for all relevant life-cycle stakeholders, including acquirers, developers, users, maintainers, and independent assessors.

The development of security measurement baselines requires a three-level definition of contents. Generic security requirements and their attributes are defined at the top of the hierarchy; these top-level factors represent the user and management views.

Then, in order to make these large-scale measures useful to project staff, each top-level factor is decomposed into subfactors that address product and process concerns. The measures for these subfactors capture the concrete attributes of the software and form the basis for the development of a comprehensive measurement model.

Because they guide the production and sustainment processes, these measures must always be meaningful to the development staff. For example, security at this level might be expressed as the system's ability to detect and recover from execution faults.

Finally, each subfactor is further decomposed into a set of base metrics, which are used to drive the actual data collection process during the various life-cycle phases. These metrics also constitute the measurement baseline, which is used to evaluate the level of assurance throughout the life cycle. The data obtained from the measures in the baseline helps decision makers estimate the current status and eventual ability of the system to satisfy all of the requirements of the system.

4.2 Assessment Technologies

The goal of software assessment technologies is to collect metric data to support the aims of the measurement process. In their general form, assessment technologies are automated utilities that facilitate such common life-cycle activities as build management and debugging. Integrated development environments (IDEs) are probably the most common commercial instances of such technologies. For example, some IDEs facilitate developers' work in the cloud using web-based tools.

By automating work that would otherwise require time consuming visual inspections, the organization can learn more about the nature of its software in a shorter period of time. Assessment technologies can also help the organization gain greater management control over its software assets by quickly and accurately predicting costs, effort, and schedules. In addition, by enabling more thorough routine analyses, automated assessment tools can help the organization develop products that better satisfy its specific security requirements. Automated tools can be used to prove the worth of any prospective new tool or technology the organization plans to adopt.

Finally, automated tools can improve the overall life-cycle process by generating and cataloging feedback from ongoing assessments.

Automated assessment tools are most frequently used for code reviews and inspections. The review program or tool typically displays a list of violations based on programming standards embedded in the tool. This list can be further investigated to determine factors such as root cause. An automated review tool can also use pre-programmed solutions to find ways to correct issues that might come up in the analysis. Examples of assessment technologies of this type are static test tools.[1] Automated assessment tools have been popular for a long time; however, they are just that—tools. A number of broad-scale and influential projects are currently underway that are likely to represent future directions in the area of automated tool support.

4.3 Standard Assurance Environments: The Object Management Group Software Assurance (SwA) Ecosystem

The OMG SwA Ecosystem is designed to provide a comprehensive basis for establishing tool support for software assurance measurement [BSI 2013]. The SwA Ecosystem describes a common framework for presenting and analyzing properties of system trustworthiness. According to the OMG, the key value of the SwA Ecosystem is that it provides "end-to-end multi-segment traceability, from code to models to evidence to arguments to security requirements to policy" [OMG 2013]. This is accomplished through the deployment of code-to-state diagrams, code-to-architecture models, and code-to conceptual-framework models [OMG 2013].

The OMG SwA Ecosystem is specifically architected to improve the level of product and process analysis and achieve greater automation of risk analysis. At the top of the analysis process, the SwA Ecosystem leverages and connects existing OMG-based conceptual models and specifications and then identifies new specifications needed to complete the framework [OMG 2013]. The OMG then provides "a technical environment where formalized claims, arguments, and evidence can be brought together with formalized and abstracted software system representations to support high automation and high fidelity analysis" [OMG 2013]. Because these initial claims and representations might be developed in a range of tools, the OMG SwA Ecosystem also provides an integrated tooling environment for managing the outputs of different tool types.

The key enabler for the measurement process is the SwA Ecosystem Infrastructure, which is an open, standards-based integrated tooling environment that dramatically reduces the cost of software assurance activities [OMG 2013]. The SwA Ecosystem Infrastructure integrates different communities—Formal Methods, Assurance Case, Reverse Engineering and Static Analysis, and Dynamic Analysis—for a comprehensive system assurance solution [OMG 2013]. The SwA Ecosystem enables tools to interoperate, introduces new vendors (because each vendor leverages parts of the tool chain), normalizes uniform common fact modeling, separates data feeds from reasoning, and ensures that proofs of correctness for auditing purposes are determined by policy arguments [OMG 2013].

[1] http://www.nicta.com.au/research/projects/goanna

The SwA Ecosystem Infrastructure is entirely shaped by International Organization for Standardization (ISO)/OMG Open Standards. Therefore, it is designed to provide fundamental improvements in analysis using various automated modeling tools, including [OMG 2013]

- Semantics of Business Vocabulary and Rules (SBVR)
- Knowledge Discovery Metamodel (KDM)
- Structure Metrics Metamodel (SMM)
- Structured Assurance Case Metamodel (SACM)
- Software Assurance Evidence Metamodel (SAEM)
- Argumentation Metamodel (ARM)

By operating in this open and highly inclusive fashion, the SwA Ecosystem Infrastructure enables industry and government to leverage and connect existing policies, practices, processes, and tools in an affordable and efficient manner [OMG 2013].

The following works in progress are projects of OMG's Systems Assurance Task Force (Sys A-PTF), and they have the potential to add much greater granularity to the measurement process: the SAEM[2] and the ARM[3] [OMG 2013].

4.4 Standard Assurance Environments: Common Weakness Enumeration (CWE) and Common Attack Pattern Enumeration and Classification (CAPEC)

The CWE is arguably the best-known catalog of generic application vulnerability characteristics. The CWE is a community-developed formal list of common software weaknesses. The CWE is the standard for weakness identification, mitigation, and prevention efforts. In addition, it aligns with and leverages the Software Assurance Metrics and Tool Evaluation (SAMATE) project's various sub-efforts [MITRE 2013a].

The CWE was created specifically to provide structure and definition for the code assessment industry. The CWE provides a formally maintained repository of vulnerabilities in source code that expose applications to security breaches. The CWE's standard list and classification of software weaknesses serves as a unifying language for discourse regarding vulnerabilities, and it provides a standard basis for assessing tools and services [MITRE 2013a].

The CWE's objective is to help shape and mature the code security assessment industry and also dramatically accelerate the use and utility of software assurance capabilities for organizations in reviewing the software systems they acquire or develop [MITRE 2013a]. The CWE's formal list of software weakness types was created to provide the community with a common language for describing software security weaknesses in architecture, design, or code. The CWE also serves as a benchmark for software security tools [MITRE 2013a].

The CWE offers a common standard for weakness identification, mitigation, and prevention efforts. The CWE formalizes this through a common language for describing software security

[2] http://www.omg.org/spec/SAEM/1.0/Beta1/

[3] http://www.omg.org/spec/ARM/1.0/Beta1/

weaknesses and a standard basis for measuring the effectiveness of the software security tools that target those weaknesses [MITRE 2013a]. To facilitate this understanding, the CWE provides a high-level dictionary view of all of its enumerated weaknesses, a classification tree view that provides access to individual weaknesses, and a purely graphical view of the classification tree that allows users to better understand individual weaknesses through their broader context and relationships [MITRE 2013a].

To create the CWE, a preliminary classification and categorization of vulnerabilities, attacks, faults, and other concepts was developed to help define common software weaknesses. The result was a working document—Preliminary List of Vulnerability Examples for Researchers (PLOVER)—that lists over 1,500 diverse, real-world examples of vulnerabilities. The vulnerabilities in PLOVER are organized within a detailed conceptual framework that currently enumerates 290 types of software weaknesses, idiosyncrasies, faults, and flaws [MITRE 2013a]. PLOVER takes real-world observed faults and flaws that already exist in code, abstracts them, and groups them into common classes representing additional vulnerabilities that could exist. PLOVER organizes these vulnerabilities in an appropriate structure to make them accessible and useful to a diverse set of audiences for a diverse set of purposes.

The CWE list and its classification tree, which was derived from PLOVER, now provide a mechanism for describing various code vulnerability assessment capabilities in terms of their coverage of the different weaknesses. The CWE contains 1,500 common vulnerabilities, but it also includes detail, breadth, and classification structure from a diverse set of other industry and academic sources and examples [MITRE 2013a]. The CWE’s comprehensive definitions and descriptions support efforts to find these common types of software security flaws in code prior to fielding.

The CAPEC is another product intended to make the concept of attack patterns actionable to a broad community [MITRE 2013b]. The CAPEC standardizes the capture and definition of common attack patterns by defining a standard schema; collecting known attack patterns into an integrated enumeration that can be expanded and enhanced by users; and classifying attack patterns so that users can easily apply the subset of the CAPEC that is most appropriate to their situation [MITRE 2013b]. The intent of the CAPEC is to standardize attack pattern definitions as a key element of a broader attempt to integrate intent into a standard set of tools and methods that includes the CWE and the Common Vulnerability Enumeration (CVE) [MITRE 2013b]. Though the CAPEC is still under development, it provides a basis for characterizing and collecting data about real-world events that are of vital interest to users.

4.5 Standard Assurance Environments: Consortium for IT Software Quality Characteristics Project

One of the main goals of the Consortium for IT Software Quality (CISQ) characteristics project is the identification of factors that induce critical programming errors. These errors result from specific architectural and/or coding bad practices that carry the highest immediate or long-term business disruption risk [CISQ 2012]. These practices are often technology related and can range from weak naming conventions to failures in national security systems. The large factors that normally determine whether an error will be produced are context, business objectives, and dependency [OMG 2012].

The chief outcome of the CISQ characteristics project is a classification of a set of primary best practices. This classification also aligns with and operationalizes the intent and purposes of ISO 9126: 2004. The CISQ classification focuses on programming practices that are likely to cause critical errors in software; such practices involve software patterns known to lead to unexpected behavior such as uninitialized variables and null pointers.

CISQ also recommends the use of explicit error management methods in the creation of procedures for performing `insert`, `update`, `delete`, `create table`, or `select` functions. The CISQ also makes recommendations for encapsulation of classes in multithread environments. For instance, external classes should not be given direct access to an object's member fields; if the system assumes that these fields cannot be changed, the potential exists for malicious code injection. Finally, the CISQ highlights the importance of overall complexity of components as a factor in program failure due to the quantitative evidence of the relationship between complexity and error frequency.

4.6 Software Productivity Research

Over the history of the profession, Software Productivity Research[4] (SPR) has maintained a repository for research on software estimation, measurement, and assessment data. That data has been used since 1983 to provide guidance for the productivity, performance, and quality of the software process.

The SPR Knowledge Base is an industry-wide asset that contains vast amounts of project data that SPR has collected and rigorously maintained over the past 30 years. Organizations can use this data to perform comprehensive comparisons to a relevant set of peers, as well as best-in-class organizations. Such comparisons enable managers to prioritize their efforts based on peer and best-in-class data. Findings from activity based on this data have created most of the fundamental assumptions about what constitutes quality and security in the production process for commercial software.

[4] http://www.spr.com/

5 Current Relevant Software Assurance Measures

5.1 Common Assurance Metrics

Over the past 50 years well known—and in many cases notable—measures and standard units of measure have been developed and utilized in the software development process. As a result, there is already a common body of measures that can be adapted to form the basis for an effective software assurance measurement process. From the standpoint of standard measurement, there are 39 commonly recognized measures that can apply to assurance, as listed in IEEE 1044. Most of these measures are based on error and defect detection, or execution failures. That includes both direct measures, which tell the analyst something about the software's structure, and indirect measurement proxies that tell the analyst how the product or process performs within a given environment.

Standard direct measures of assurance can include such measures as mean time to failure and residual fault count. Indirect measurement proxies include estimates of complexity, design structure, and software release readiness. Successful application of these measures obviously depends on their use in the specified environment. However, these are commonly accepted measures whose outcomes characterize issues that are well understood.

5.2 Direct and Indirect Measures of Software Performance

As noted earlier, software assurance measures fall into two generic categories: direct observations of behavior and indirect proxies. With direct observations, the only goal of the measurement is to ensure that the behavioral outcomes are consistently and accurately recorded regardless of the type of system being analyzed. Indirect assurance proxies are a bit more complicated. With assurance proxies there must be a way to infer assurance value from what can be observed.

In order to establish the relationship between performance and assurance, it must be possible to deduce something about the software's internal functioning and structure from externally observable phenomena. Factors useful in supporting observations of this type are complexity, object oriented characteristics, temporal constraints (e.g., processing demands), interrupt and error handling conditions, and exception handling; data from these measures must be easily obtainable and useful to various stakeholders.

In the following paragraphs, we describe the most commonly used measures of system assurance. The list is not meant to be exhaustive and is provided for the sake of illustration. The measures discussed here have traditionally supported conclusions about the level of assurance of the system from externally observable fact.

The most fundamental measure of product and process correctness is probably defect density. Defect density is a direct measure of performance that allows evaluators to make inferences about the state of system assurance based on the number of confirmed defects detected in a software component. To be meaningful, defect density must be measured during a defined period of development or operation. The defect count is divided by the size of the software/component, and defects must be confirmed as present for the defect density estimate to be considered valid. The period can be expressed as duration or phase. Size is measured in function points (FPs) or source lines of code (SLOC). Defect density is calculated as number of SLOC or FP divided by the defect count.

Defect density is primarily used to support decision making about product and process performance. It indicates the relative number of defects in a set of software components so that high-risk components can be identified and resources can be focused on assuring them. Defect density is also used to compare a set of products to quantify the security and quality of each product and improve products with low quality. Because defect identification is involved, the analyses that generate a defect density estimate normally take place after a design or code inspection. If the defect density is outside a given range after several inspections, there may be problems in the inspection process as well as the product. This makes defect density a particularly useful index of performance.

A cumulative failure profile (CFP) can be used to infer the level of assurance of a given artifact after initial testing takes place. A CFP generally reflects the failure rate of a system at a given point in the testing process. The failure rate is usually characterized by time units and varies over the life cycle of the system. The failure rate is the total number of failures within an item population divided by the particular measurement interval. Thus, failure rate can be used to track life-cycle performance. For example, a software artifact's failure rate in its fifth year of service may be many times greater than its failure rate during its first year of service. Therefore, defects can be graphed over time; based on that information, activities like additional testing can be prescribed.

Halstead's Software Science is a good example of an inferential metric. It is based on complexity. Halstead's metric can be derived from the code after the design phase is completed and the code is generated. The computations are based on the numbers of distinct operators and operands, and the total numbers of operators and operands in a program. By using these measures, Halstead derived a fairly reliable index of program length and complexity. Since complexity is associated with program defects, this index is an indirect measure of assurance.

However, Halstead's Software Science is not the primary method used to obtain a mathematical estimate of product status. That distinction goes to McCabe's Complexity Metric, which is a metric similar to Halstead's except that calculations are based on the control flow embedded in the program. Branches and loops are used to calculate cyclomatic complexity, and that calculation is used to estimate potential complexity of the system. This information is particularly useful for estimating assurance prior to the actual coding phase since it predicts the effort and rigor that would be required to build and test a system based on its estimated cyclomatic complexity.

5.3 The Other Side of the Equation: Problems with Measures of Software Size

Counting of lines of code is perhaps the first way the software industry measured itself. The number of lines of code is a measure that can serve as a proxy for a number of important concerns, including program size and programming effort. When used in a standard fashion—such as thousand lines of code (KLOC)—it forms the basis for estimating resource needs. Size is an inherent characteristic of a piece of software just like weight is an inherent characteristic of a tangible material. Software size, as a standard measure of resource commitment, must be considered along with product and performance measurement.

SLOC has traditionally been a measure for characterizing productivity. The real advantage of lines of code is that the concept is easily understandable to conventional managers. However, a lines-of-code measurement is not as simple and accurate as it might seem. In order to accurately describe software in terms of lines of code, code must be measured in terms of its *technical* size and *functional* size. Technical size is the programming itself. Functional size includes all of the additional features such as database software, data entry software, and incident handling software.

The most common technical sizing method is still probably number of lines of code per software object. However, function point analysis (FPA) has become the standard for software size estimation in the industry. FPA measures the size of the software deliverable from a user's perspective. Function point sizing is based on user requirements and provides an accurate representation of both size for the developer/estimator and business functionality being delivered to the customer. FPA has a history of statistical accuracy and has been used extensively in application development management (ADM) or outsourcing engagements. In that respect, FPA serves as a global standard for delivering services and measuring performance.

FPA includes the identification and weighting of user recognizable inputs, outputs, and data stores. The size value is then available for use in conjunction with numerous measures to quantify and evaluate software delivery and performance. Typical management indexes that are based on function points include

- development cost per function point
- delivered defects per function point
- function points per staff month

The primary advantages of FPA are that it can be applied early in the software development life cycle, and it is not dependent on lines of code. In effect, FPA is technology agnostic and can be used for comparative analysis across organizations and industries. Since the inception of FPA, several variations have evolved, and the family of functional sizing techniques has broadened to include such sizing methods as Use Case Points, FP Lite, Early and Quick FPs, and Story Points, as well as variations developed by the Common Software Measurement International Consortium (COSMIC) and the Netherlands Software Metrics Association (NESMA).

6 Summary and Conclusions

Measurement is an important part of assurance because it provides the basis for understanding software work. Common sense tells us that any complicated activity can be better managed if its elements are known and their interrelationships are clearly understood. Conversely, we can also assume that exploitable defects in software are the result of haphazard and inconsistent knowledge of the work. Organizations must be able to see and control the actions that underlie projects to assure that the work is done correctly. Therefore, it is reasonable to conclude that the continuing presence of a large number of defects in software indicates that we need greater visibility into the product development and sustainment processes. In any industry systematic visibility comes through formal measurement.

America’s software systems are insecure. How pervasive is the problem? A major software security firm found that “58 percent of all applications across supplier types [failed] to meet acceptable levels of security” [Veracode 2009]. The root cause of the problem is that software development and sustainment processes are so complex that it is difficult for managers to assure correct practice. Therefore, the first order of business in leveraging better product security is to create a reliable monitoring capability that allows managers to directly oversee the projects they supervise.

Software projects have always been risky propositions. The production of a software system involves the translation of a customer’s abstract ideas into specific programmed behaviors. That abstract quality makes it difficult to ensure a consistent and repeatable outcome. Software project statistics reflect this difficulty. A series of annual “Chaos” surveys over the past decade found that 32 percent of all projects were delivered according to time, budget, and functionality requirements; 44 percent were late, over budget, or delivered with less than the required features and functions; and 24 percent were cancelled prior to completion or delivered and never used [Standish 2009]. A 2005 survey had similar findings: Approximately 33 percent of all projects were cancelled prior to deployment, 75 percent of all projects were completed late, and nearly 50 percent lacked required features and functions [Borland 2005]. Capers Jones has consistently found that, depending on project size, between 25 and 60 percent of all projects fail, where “failure” means that the project is canceled or grossly exceeds its schedule estimates [Jones 1994].

But the traditional problems with software product quality and productivity are just the starting point. When product defects were simply a quality concern, the occasional bug had ramifications for marketing and customer relations. Today, adversaries can exploit program defects, causing disastrous outcomes. Therefore, excuses for software product defects are no longer acceptable.

Assurance starts with understanding. So it is critical that the activities that take place in the life cycle are adequately understood. The most telling result of an earlier study was the impact of lack of knowledge on software project success. Between 44 and 48 percent of project failures were caused by developers who did not have a clear understanding of the activities being performed in their own projects. The most common causes of failure were a lack of project management, a project manager’s lack of skill, and a project manager’s inability to monitor project activity [Tech Republic 2009].

6.1 Business Realities Versus Due Care

It is difficult to make a business case for taking the additional steps to manage the life cycle. Extra steps to ensure better visibility into projects cut into profitability or add to the cost of the product (at least in the short term). Because purchase decisions are based on price, not intangible qualities like security, developers have no incentive to go the extra mile to ensure that their overall process is suitably organized and managed.

Software developers can still take a practical approach to ensure correct products that are safe from exploitation. Over the past 20 years, an array of authoritative studies has supported the idea that the systematic assurance of software work will reap two specific rewards. The direct benefits of systematic assurance are that production will be more cost efficient and overall product quality will be better. At the same time, increasing the visibility of the organization’s development, sustainment, and acquisition processes will lead to fewer original mistakes and less costly rework. Experts have estimated that rework represents the predominant cost of a software product, so the preceding factors are powerful arguments for any effort to increase the across-the-board monitoring and control capability of a software operation [Jones 2005].

6.2 Formal Measurement – Better Assurance

Formal measurement is important because, throughout the history of the profession, software organizations have tended to fail because of the effects of the “dark star” of process entropy. Process entropy is a mixture of competitive pressure, mind-numbing technical evolution, and ever-increasing project scale. Because software product development is creative, the continuous appearance of unknowns must be approached intuitively. And, because the technical staff’s methods are typically personal and individual, an approach that is tremendously effective on one project might not apply to the next [Humphrey 1994]. Given the resource and scheduling problems such variation causes, an effective organization will use some form of controlled monitoring practice to ensure consistent execution of project tasks. The organization that follows a disciplined set of monitoring practices can duplicate its successes as well as learn from its failures. Disciplined execution standardizes the outcomes of the process, making them comparable across projects.

Measurement allows software products and processes to be understood in concrete, comparative terms so that they can be managed at all levels in the organization. Because software is intangible, formal measurement is the only basis for assuring disciplined execution of the software life cycle. Measurement is particularly useful in the case of managing complex systems where the requirement for integration places exceptional demands for coordination of the process. Moreover, situations such as multi-tiered vendor arrangements and interoperable product lines can be coordinated and controlled only through objective understanding of the situation. By defining the best practices for measuring each life-cycle stage, defects can be minimized at all levels of work. Formal measurement also helps ensure early risk detection and true product-in-process visibility. That visibility helps control the number of exploitable defects in code, which in its most extended ramification means a much brighter future for a technical society like ours.

Appendix A: Common Measures by Life-Cycle Area

Common Productivity Measures

These measures can be collected throughout the life cycle, plotted, and analyzed to identify process activities that are most prone to errors, suggest steps to prevent recurrence of similar errors, suggest procedures for earlier detection of faults, and make improvements to the development process:

- time spent—hours, total hours less slack
- elapsed time—hours
- staff hours—by type
- staff months—by type
- staff years—by type

Requirements Measures

Requirements size involves a simple count. Large components are assumed to have a larger number of residual errors; thus, their reliability may be affected.

- number of pages or words
- number of requirements
- number of functions

Requirements traceability is computed using the equation `RT = R1/R2 x 100%`, where R1 is the number of requirements met by the design, and R2 is the number of original requirements. Traceability identifies requirements missing from or added to the design.

Completeness is used to assess the correctness of coverage of the specification.

- count—total defined functions
- count—functions satisfactorily defined
- count—functions not satisfactorily defined
- count—functions used in final product
- count—functions not used in final product
- count—functions referenced by other functions
- count—data references having an origin
- count—data references satisfactorily defined

Fault days are the days from creation to removal that a fault spends in a software product.

- The fault days, FDi, are the days from the creation of the fault to its removal (FD = sum of FDi).
- This measure is an indicator of the effectiveness of the software design and software development process.

- A high value may indicate untimely removal of faults and/or existence of many faults due to an ineffective software development process.

Design Measures

These measures provide an early indication of project status, enable selection of alternative designs, identify potential problems early in the software development process, limit complexity, and help in deciding how to modularize so that the resulting modules are both testable and maintainable.

Complexity measures identify modules that are complex or difficult to test.

- number of parameters per module
- number of states or data partitions per parameter
- number of branches in each module

Defect density is used after design inspections or modifications to assess the inspection process.

- total number of unique defects detected during the inspection activity
- total number of unique defects detected during the software development activity
- total number of inspections to date
- number of source lines of design statements in thousands (KSLOD)

Design size is used to estimate the size of the software design or software design documents.

- number of pages or words
- DLOC (lines of Program Design Language)
- number of modules
- number of functions
- number of inputs and outputs
- number of interfaces

Estimated number of modules provides a measure of *product size*, against which the completeness of subsequent module-based activities can be assessed.

- The estimate for the number of modules is given by `NM = S/M` where S is the estimated size in LOC, and M is the median module size found in similar projects.
- The estimate NM can be compared to the median number of modules for other projects.

Module fault measures identify fault-prone modules.

- number of faults associated with each module
- number of requirements faults detected during detailed design
- number of structural design faults detected during detailed design

Structural measures (fan-in / fan-out) represent the number of modules that call/are called by a given module. This measure identifies whether

- the system decomposition is adequate
- there are no modules that cause bottlenecks

- there are no missing levels in the hierarchical decomposition
- there are no unused modules (“dead” code)

Code Measures

Lines of code (LOC) is an indication of size, which allows for estimation of effort, time scale, and total number of faults for the application.

- Length of a program partly depends on the language the code is written in, thus making comparison using LOC difficult.
- LOC can be a useful measure if the projects being compared are consistent in their development methods.
- The advantages of LOC are that it is conceptually simple, easily automated, and inexpensive.
- Because of its disadvantages, the use of LOC as a management metric (e.g., for project sizing beginning from the software requirements activity) is controversial.
- There are uses for this metric in error analysis, such as to estimate the values of other measures.

Number of entries/exits/modules is used to assess the complexity of an architecture by counting the number of entry and exit points in each module.

- The equation to determine the measure for the `ith` module is simply $M_i = e_i + x_i$, where e_i is the number of entry points for the `ith` module, and x_i is the number of exit points for the `ith` module.

Cyclomatic complexity is used to determine the structural complexity of a coded module in order to limit its complexity, which can lead to a high number of defects and maintenance costs. It is also used to identify the minimum number of test paths to assure test coverage.

Amount of data is defined by the number of inputs/outputs and or by the number of variables.

Live variables are the variables whose values could change during execution.

- The average number of live variables per line of code is the sum of the number of live variables for each line, divided by the number of lines of code.

Variable scope is the number of source statements between the first and last reference of the variable.

- With large scopes, the understandability and readability of the code is reduced.

Variable span is the number of source statements between successive references of the variable. With large spans, it is more likely that a far-back reference will be forgotten.

Test Measures

Common test measures include

- number of software integration test cases planned/executed involving each module
- number of black box test cases planned/executed per module
- number of requirements faults detected (which also reassesses the quality of the specification)

- total number of planned white/black box test cases run to completion
- number of planned software integration tests run to completion
- number of unplanned test cases required during test activity

Statement coverage measures the percentage of statements executed (to assure that each statement has been tested at least once).

Branch coverage measures the percentage of branches executed.

Path coverage measures the percentage of program paths executed.

- It is generally impractical and inefficient to test all the paths in a program.
- The count of the number of paths may be reduced by treating all possible loop iterations as one path.
- Path coverage may be used to ensure 100 percent coverage of critical (safety or security related) paths.

Data flow coverage measures the definition and use of variables and data structures.

Test coverage measures the completeness of the testing activity.

- Test coverage is the percent of test cases or functional capabilities multiplied by the percentage of segments, statements, branches, or path test results.

Sustainment Measures

Common change measures include

- number of changes
- cost/effort of changes
- time required for each change
- LOC added, deleted, or modified
- number of fixes or enhancements

Customer ratings are based on results of customer surveys, ratings, or satisfaction scores of a vendor’s product or customer services.

Customer service measures are based on results of customer surveys, ratings, or satisfaction scores of a vendor’s service staff.

- number of hotline calls received
- number of fixes for each type of product
- number of hours required for fixes
- number of hours for training (for each type of product)

Appendix B: Common Measures by Type

Direct Defect/Error/Fault Measures

The following is a list of some measures of direct defects, errors, and faults.

- number of unresolved faults at planned end of activity
- number of faults that have not been corrected
- number of outstanding change requests
- number of software requirements faults detected during reviews and walkthroughs
- number of software design faults detected during reviews and walkthroughs
- number of requirement, design, coding faults found during unit testing
- number of requirement, design, coding faults found during integration testing
- number of revisions, additions, deletions, or modifications of code per module
- number of requests to change the software requirements specification
- number of requests to change the software design
- age of open real problem reports
- age of unevaluated problem reports
- age of real closed problem reports
- point in process when error(s) discovered
- rate of error discovery in defects/hour
- number of faults detected per module
- number of errors by type (e.g., logic, computational, interface, documentation)
- number of errors by cause or origin
- number of errors by severity (e.g., critical, major, cosmetic)

Proxy Defect/Error/Fault Measures

Fault density is computed by dividing the number of faults by the size (usually in KLOC). Fault density can be used to perform the following:

- predict remaining faults by comparison with expected fault density
- determine if sufficient testing has been done based on predetermined goals
- establish standard fault densities for comparison and prediction

Defect age is the time between when a defect is introduced and when it is detected or fixed.

Defect response time is the time between when a defect is detected and when it is closed.

Defect cost is a sum of the costs to analyze and fix the defect and the cost of failures already incurred due to the defect.

The *defect removal efficiency* (DRE) is the percentage of defects removed during an activity. The DRE can also be computed for each software development activity and plotted on a bar graph to show the relative defect removal efficiencies for each activity. Alternatively, the DRE may be computed for a specific task or technique (e.g., design inspection, code walkthrough, unit test, six-month operation).

Mean time to failure estimates the mean time to the next failure by computing the average of all failure times.

- This metric is the basic parameter in most reliability models. High values imply good reliability.

Failure rate indicates growth in software reliability as a function of test time.

- This metric requires two primitives: t_i, the observed time between failures for a given severity level i, and f_i, the number of failures of a given severity level in the ith time interval.

Cumulative failure profile uses a graphical technique to predict reliability, estimate additional testing time needed to reach an acceptable reliability level, and identify modules and subsystems that require additional testing.

- This metric requires one primitive, f_i, the total number of failures of a given severity level i in a given time interval.
- Cumulative failures are plotted on a time scale. The shape of the curve is used to project when testing will be complete, and to assess reliability.
- This metric can provide an indication of clustering of faults in modules, suggesting further testing for these modules.

Appendix C: Common Measurement Standards

The following IEEE standards define measures for software:

- IEEE 1044: IEEE Standard Classification for Software Anomalies (2009)
- IEEE 1045: IEEE Standard for Software Productivity Metrics (1992)
- IEEE 982.1 Standard Dictionary of Measures of the Software Aspects of Dependability (2005)
- IEEE 982.2 Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software (1988)
- IEEE 1061: IEEE Standard for a Software Quality Metrics Methodology (1998)
- ISO/IEC 25010—Systems and software engineering—Systems and Software Quality Requirements and Evaluation

Characterization of Defects: IEEE 1044

Since most of software assurance is oriented toward identifying and reporting defects, there is a need for a fundamental definition of bugs. IEEE 1044.1 2005 fulfills this role. It provides a standard classification of software anomalies commonly found in software and its documentation. Its companion standard—IEEE 1044.1—describes the types of anomalies found during any software life-cycle phase. It also provides a comprehensive list of software anomaly classifications and related data items that are helpful in the identification and tracking of anomalies.

This standard is not intended as a definition of procedural or format requirements. It *does* identify some classification measures and non-exhaustively defines the data supporting the analysis of an anomaly. This standard is applicable to any software during any phase of the life cycle including

- critical systems software
- commercial applications
- system software
- support software
- testware
- firmware

Productivity Measures: IEEE 1045

This standard categorizes measurement primitives by attributes. An attribute is a measurable characteristic of a primitive. This standard does *not* claim to improve productivity, only to measure it. It provides a consistent and measurement-based way to characterize software productivity. Software productivity measurement terminology is meant to ensure an understanding of the basis for measuring both code and document production in a project. It should be noted that although this standard describes measurements to characterize the software process it does *not* measure the quality of software.

Software Reliability: IEEE 982

The objective of this standard is to provide the software community with defined measures currently used as indicators of reliability. The predictive nature of these measures is dependent on correct application of a valid model. This standard presents

- a selection of applicable measures
- the proper conditions for using each measure
- the methods of computation
- a framework for a common language among users

In addition it provides

- the means for continual assessment of the process and product
- the stipulation of a teaching phase in the life cycle

By emphasizing early reliability assessment, this standard also supports measurement-based methods for improving product reliability.

Software Quality Metrics Methodology IEEE 1061

Although there are disagreements in terminology, IEEE 1061 is generally considered to provide an outline of the methodology to be specified in the ISO/IEC 9126 standard—Software Product Quality Characteristics. ISO 9126 is concerned primarily with defining the security characteristics used in the evaluation of software products. It provides the definition of those characteristics. In addition, it provides an associated security evaluation process to be used when specifying the requirements for and evaluating the security of software products throughout their life cycle. It sets out six security characteristics, which are intended to be exhaustive.

The development process requires user and developer to understand the same security characteristics, since they apply to both requirements and acceptance. Developers are interested in intermediate product security as well as final product security. That is because they are responsible for producing software that satisfies security requirements. Therefore, in order to evaluate the intermediate product security at each phase of the development cycle, developers must use different measures for the same characteristics. That is because the same measures are not applicable to all phases of the life cycle. For instance, unimplemented user requirements cannot be measured until the post-coding phase; moreover, qualitative factors such as design coherence are meaningful only at the pre-coding stage. By the time the software gets to the testing level, it is too late to address runtime errors, so a measure of runtime errors must be applied where it can do some good (e.g., at coding).

A manager is typically more interested in overall security rather than in a specific security characteristic. For this reason he/she must assign weights, reflecting business requirements, to individual characteristics. The manager may also need to balance the security improvement with management criteria such as schedule delay or cost overrun because he/she wishes to optimize security within limited cost, human resources, and time frame.

Systems and Software Quality Requirements and Evaluation (SQuaRE) [ISO/IEC 25010:2011]

ISO 25010 helps eliminate misunderstanding between purchaser and supplier. The purchaser is able to understand clearly and communicate his/her requirements for the product to be developed. The supplier is able to understand the requirement, and is able to assess with confidence whether it is possible to provide the product with the right level of software security. The six characteristics proposed by the ISO 25010 security model are

- functionality
- assurance
- usability
- efficiency
- maintainability
- portability

These characteristics are then decomposed into sub-characteristics and measures.

Appendix D Automated Code Checking Tools

The following list is adapted from Wikipedia’s list of tools for static code analysis (http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis). This list provides examples of tools and is not a recommendation, nor is it intended to be exhaustive or prescriptive.

Multi-Language

Axivion Bauhaus Suite—A tool for Ada, C, C++, C#, and Java code that comprises various analyses such as architecture checking, interface analyses, and clone detection.

Black Duck Suite—Analyzes the composition of software source code and binary files, searches for reusable code, manages open source and third-party code approval, honors the legal obligations associated with mixed-origin code, and monitors related security vulnerabilities.

BugScout—Detects security flaws in Java, PHP, ASP, and C# web applications.

CAST Application Intelligence Platform—Detailed, audience-specific dashboards to measure quality and productivity. 30+ languages, C/C++, Java, .NET, Oracle, PeopleSoft, SAP, Siebel, Spring, Struts, Hibernate, and all major databases.

ChecKing—Integrated software quality portal that allows users to manage the quality of all phases of software development. It includes static code analyzers for Java, JSP, Javascript, HTML, XML, .NET (C#, ASP.NET, VB.NET, etc.), PL/SQL, embedded SQL, SAP ABAP IV, Natural/Adabas, C/C++, COBOL, JCL, and PowerBuilder.

Checkmarx—Allows security professionals and software developers to detect software security vulnerabilities and business logic flaws in Java, C# /.NET, ASP.NET, PHP, C/C++, ASP, Android Java, Objective C (iOS), Visual Basic 6, JavaScript, Ruby, Python, Perl, PL/SQL, APEX, and HTML5.

Coverity SAVE—Coverity® Static Analysis Verification Engine (Coverity SAVE™) is a static code analysis tool for C, C++, C#, and Java source code. Coverity commercialized a research tool for finding bugs through static analysis, the Stanford Checker, which used abstract interpretation to identify defects in source code.

DMS Software Reengineering Toolkit—Supports custom analysis of C, C++, C#, Java, COBOL, PHP, Visual Basic, and many other languages. Also COTS tools for clone analysis, dead code analysis, and style checking.

HP Fortify Source Code Analyzer—Helps developers identify software security vulnerabilities in C/C++, Java, JSP, .NET, ASP.NET, ColdFusion, classic ASP, PHP, Visual Basic 6, VBScript, JavaScript, PL/SQL, T-SQL, Python and COBOL, and configuration files.

GrammaTech CodeSonar—Defect detection (Buffer overruns, memory leaks), concurrency and security checks, architecture visualization, and software measures for C, C++, and Java source code.

IBM Rational AppScan Source Edition—Analyzes source code to identify security vulnerabilities while integrating security testing with software development processes and systems; supports C/C++, .NET, Java, JSP, JavaScript, ColdFusion, Classic ASP, PHP, Perl, VisualBasic 6, PL/SQL, T-SQL, and COBOL

Imagix 4D—Identifies problems in variable use, task interaction, and concurrency, especially in embedded applications, as part of an overall system for understanding, improving, and documenting C, C++, and Java code.

Klocwork Insight—Provides security vulnerability, defect detection and build-over-build trend analysis for C, C++, C#, and Java.

LDRA Testbed—A software analysis and testing tool suite for C, C++, Ada83, Ada95, and Assembler (Intel, Freescale, Texas Instruments).

MALPAS—A software static analysis toolset for a variety of languages including Ada, C, Pascal, and Assembler (Intel, PowerPC, and Motorola). Used primarily for safety critical applications in nuclear and aerospace industries.

Moose—Moose started as a software analysis platform with many tools to manipulate, assess or visualize software. It can evolve to a more generic data analysis platform. Supported languages are C/C++, Java, Smalltalk, and .NET. More may be added.

Parasoft—Provides static analysis (pattern-based, flow-based, in-line measures) for C, C++, Java, .NET (C#, VB.NET, etc.), JSP, JavaScript, XML, and other languages. Through a development testing platform, static code analysis functionality is integrated with unit testing, peer code review, runtime error detection, and traceability.

Copy/Paste Detector (CPD)—PMDs duplicate code detection (e.g., for Java, JSP, C, C++, ColdFusion, PHP, and JavaScript code).

Polyspace—Uses abstract interpretation to detect and prove the absence of certain runtime errors in source code for C, C++, and Ada.

Protecode—Analyzes the composition of software source code and binary files, searches for open source and third party code and their associated licensing obligations. (Protecode can also detect security vulnerabilities.)

ResourceMiner—Details multipurpose analysis and measures, allows users to develop their own rules for masschange and generator development. Supports 30+ legacy and modern languages and all major databases.

Semmle—Supports Java, C, C++, and C#.

SofCheck Inspector—Static detection of logic errors, race conditions, and redundant code for Ada and Java; automatically extracts pre/post-conditions from code.

Sonar—A continuous inspection engine to manage technical debt: unit tests, complexity, duplication, design, comments, coding standards, and potential problems. Supported languages include ABAP, C, COBOL, C#, Flex, Forms, Groovy, Java, JavaScript, Natural, PHP, PL/SQL, Visual Basic 6, Web, XML, and Python.

Sotoarc/Sotograph—Architecture and quality in-depth analysis and monitoring for C, C++, C#, and Java.

SQuORE—A multi-purpose and multi-language monitoring tool for software projects.

Understand—Analyzes Ada, C, C++, C#, COBOL, CSS, Delphi, Fortran, HTML, Java, JavaScript, Jovial, Pascal, PHP, PL/M, Python, VHDL, and XML—reverse engineering of source, code navigation, and measurement tool.

Veracode—Finds security flaws in application binaries and byte code without requiring source. Supported languages include C, C++, .NET (C#, C++/CLI, VB.NET, ASP.NET), Java, JSP, ColdFusion, PHP, Ruby on Rails, and Objective-C, including mobile applications on the Windows Mobile, BlackBerry, Android, and iOS platforms.

Visual Studio Team System—Analyzes C++, C# source codes. Visual Studio Team system is only available in the team suite and development edition.

Yasca—Yet Another Source Code Analyzer, a plugin-based framework to scan arbitrary file types, with plugins for C/C++, Java, JavaScript, ASP, PHP, HTML/CSS, ColdFusion, COBOL, and other file types. It integrates with other scanners, including FindBugs, PMD, and Pixy.

.NET

CodeIt.Right—Combines static code analysis and automatic refactoring to best practices, which allows automatic correction of code errors and violations; supports C# and VB.NET.

CodeRush—A plugin for Visual Studio, it addresses a multitude of shortcomings with the popular IDE. CodeRush alerts users to violations of best practices by using static code analysis.

FxCop—Free static analysis for Microsoft .NET programs that compile to CIL. FxCop, a standalone tool developed by Microsoft, is integrated in some Microsoft Visual Studio editions.

Kalistick—Mixing from the Cloud: static code analysis with best practice tips and collaborative tools for agile teams.

NDepend—Simplifies managing a complex .NET code base by analyzing and visualizing code dependencies, defining design rules, doing impact analysis, and comparing different versions of the code. (NDepend integrates into Visual Studio.)

Parasoft dotTEST—A static analysis, unit testing, and code review plugin for Visual Studio; works with languages for Microsoft .NET Framework and .NET Compact Framework, including C#, VB.NET, ASP.NET, and Managed C++.

StyleCop—Analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Microsoft Visual Studio or integrated into an MSBuild project. StyleCop is available as a free download from Microsoft.

ActionScript

Apparat—A language manipulation and optimization framework consisting of intermediate representations for ActionScript.

Ada

AdaControl—A tool to control occurrences of various entities or programming patterns in Ada code; AdaControl is used for checking coding standards, enforcing safety related rules, and supporting various manual inspections.

Fluctuat—Abstract interpreter for the validation of numerical properties of programs.

LDRA Testbed—A software analysis and testing tool suite for Ada83/95.

Polyspace—Uses abstract interpretation to detect and prove the absence of certain runtime errors in source code.

SofCheck Inspector—(Bought by AdaCore) Static detection of logic errors, race conditions, and redundant code for Ada; automatically extracts pre/post-conditions from code.

C/C++

Astrée—exhaustive search for runtime errors and assertion violations by abstract interpretation; tailored towards critical code (avionics).

BLAST (Berkeley Lazy Abstraction Software Verification Tool)—A software model checker for C programs based on lazy abstraction.

Cppcheck—An open source tool that checks for several types of errors, including use of STL.

Cpplint—An open source tool that checks for compliance with Google's style guide for C++ coding.

Clang—A compiler that includes a static analyzer.

Coccinelle—Source code pattern matching and transformation.

ECLAIR—A platform for the automatic analysis, verification, testing, and transformation of C and C++ programs.

Eclipse (software)—An IDE that includes a static code analyzer (CODAN).

Fluctuat—An abstract interpreter for the validation of numerical properties of programs.

Frama-C—A static analysis framework for C.

Goanna—A software analysis tool for C/C++.

GrammaTech CodeSonar—Defect detection (buffer overruns, memory leaks), concurrency and security checks, architecture visualization and software measures for C, C++, and Java source code.

Lint—The original static code analyzer for C.

LDRA Testbed—A software analysis and testing tool suite for C/C++.

Makedepend—A UNIX tool to generate dependencies of C source files.

Parasoft C/C++test—A C/C++ tool that does static analysis, unit testing, code review, and runtime error detection; plugins are available for Visual Studio and Eclipse-based IDEs.

PC-Lint—A software analysis tool for C/C++.

Polyspace—Uses abstract interpretation to detect and prove the absence of certain runtime errors in source code.

PVS-Studio—A software analysis tool for C, C++, C++11, and C++/CX (Component Extensions).

PRQA QA·C and QA·C++—Deep static analysis of C/C++ for quality assurance and guideline/coding standard enforcement.

SLAM project—A project of Microsoft Research for checking that software satisfies critical behavioral properties of the interfaces it uses.

Sparse—A tool designed to find faults in the Linux kernel.

Splint—An open source evolved version of Lint, for C.

Java

AgileJ StructureViews—Reverse engineered Java class diagrams with an emphasis on filtering.

ObjectWeb ASM—Allows decomposing, modifying, and recomposing binary Java classes (i.e., byte code).

Checkstyle—Besides some static code analysis, Checkstyle can be used to show violations of a configured coding standard.

FindBugs—An open source static bytecode analyzer for Java (based on Jakarta BCEL) from the University of Maryland.

GrammaTech CodeSonar—Defect detection (Buffer overruns, memory leaks), concurrency and security checks, architecture visualization and software measures for C, C++, and Java source code.

Hammurapi—A versatile code review program; Hammurapi is free for non-commercial use.

Jtest—A testing and static code analysis product by Parasoft.

Kalistick—A cloud-based platform to manage and optimize code quality for Agile teams with DevOps spirit.

LDRA Testbed—A software analysis and testing tool suite for Java.

PMD—A static rule-set-based Java source code analyzer that identifies potential problems.

SemmleCode—Object oriented code queries for static program analysis.

SonarJ—Monitors conformance of code to intended architecture, also computes a wide range of software measures.

Soot—A language manipulation and optimization framework consisting of intermediate languages for Java.

Squale—A platform to manage software quality (also available for other languages, using commercial analysis tools).

IntelliJ IDEA—A cross-platform Java IDE with its own set of several hundred code inspections available for analyzing code on-the-fly in the editor and bulk analysis of the whole project.

JavaScript

Closure Compiler—A JavaScript optimizer that rewrites code to be faster and smaller and checks the use of native JavaScript functions.

JSLint—A JavaScript syntax checker and validator.

JSHint—A community-driven fork of JSLint.

Objective-C

Clang—A free project that includes a static analyzer. As of version 3.2, this analyzer is included in Xcode.

Opa

Opa includes its own static analyzer. Since the language is intended for web application development, the strongly statically typed compiler checks the validity of high-level types for web data, and prevents vulnerabilities by default such as XSS attacks and database code injections.

Packages

Lintian—Checks Debian software packages for common inconsistencies and errors.

Rpmlint—Checks for common problems in rpm packages.

Perl

Perl::Critic—A tool to help enforce common Perl best practices. Most of the best practices embedded here are based on the book *Perl Best Practices* by Damian Conway.

PerlTidy—A program that act as a syntax checker and tester/enforcer for coding practices in Perl.

Padre—An IDE for Perl that also provides static code analysis to check for common beginner errors.

PHP

Mondrian—A set of open source CLI tools for analyzing and refactoring OOP PHP source code.

Python

Pychecker—A source code checking tool.

Pylint—A static code analyzer.

Formal Methods Tools

The following tools use a formal methods approach to static analysis (e.g., using static program assertions):

ECLAIR—Uses formal methods-based static code analysis techniques such as abstract interpretation and model checking combined with constraint satisfaction techniques to detect or prove the absence of certain runtime errors in source code.

ESC/Java and ESC/Java2—Based on Java Modeling Language, an enriched version of Java.

MALPAS—A formal methods tool that uses directed graphs and regular algebra to prove that software under analysis correctly meets its mathematical specification.

Polyspace—Uses abstract interpretation, a formal methods based technique to detect and prove the absence of certain runtime errors in source code for C/C++, and Ada.

SofCheck Inspector—Statically determines and document pre- and post-conditions for Java methods; statically checks preconditions at all call sites; also supports Ada.

SPARK Toolset including the SPARK Examiner—Based on the SPARK language, a subset of Ada.

Appendix E: Useful Standards for Measurement Processes

The following standards are useful for developing and maintaining measurement processes:

1. IEEE 610-1991—Standard Glossary of Software Engineering Terminology (4 parts)
2. IEEE 730 1998 Software Quality Assurance Plans
3. IEEE 730.1 1995 Software Quality Assurance Planning
4. IEEE 828 2012 Software Configuration Management Plan
5. IEEE 829 2008 Software Test Documentation
6. IEEE 1008 1997 Standard for Software Unit Testing
7. IEEE 1012 1998 Software Validation and Verification Plan
8. IEEE 1012a-1998 Content Map to IEEE/EIA 12207.1-1997
9. IEEE 1028-2008 Standard for Software Reviews
10. IEEE 1042-1987 Configuration Management Plan Guideline
11. IEEE 1045-1992 IEEE Standard for Software Productivity Metrics
12. IEEE 1059 1993 Guideline for SVV Planning (withdrawn but a good reference on testing)
13. IEEE 1063-2001, Software User Documentation
14. IEEE 1074-2006 Standard for Developing a Software Project
15. ISO/IEC 9126—2001 Software Product Evaluation—Quality Characteristics and Guidelines for Their Use
16. ISO/IEEE 12207 -2008 02-01 — Systems and Software Lifecycle Processes
17. Information Systems Audit and Control Association, Framework, COBIT (v5)
18. ISO/IEC 15408 – The Common Criteria (parts 1-3)

References

[Borland 2005]

Borland Software Corporation. “Software Delivery Optimization Maximizing the Business Value of Software.” *Borland Vision and Strategy Solution* (2005).
http://www.danysoft.com/free/bsdowp.pdf

[BSI 2013]

Build-Security-In. “The OMG Software Ecosystem, Turning Challenges into Solutions.” *Software Measurement: Its Estimation and Metrics Used* (2013).
https://buildsecurityin.us-cert.gov/swa/ecosystem.html

[CISQ 2012]

Consortium for IT Software Quality Characteristics. *Software Measurement: Its Estimation and Metrics Used* (December 3, 2012). http://it-cisq.org/software-meausrement-estimation-metrics/

[CNSS 2010]

Committee on National Security Systems. *National Information Assurance Glossary* (April 26, 2010). https://www.cnss.gov/Assets/pdf/cnssi_4009.pdf

[Fenton 1998]

Fenton, N. & Pfleeger, S. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 1998.

[Humphrey 1994]

Humphrey Watts S. *Managing the Software Process*. Addison-Wesley: Reading, MA, 1994.
http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=30884

[IATAC 2009]

Information Assurance Technology and Assurance Center (IATAC), Measuring Cyber Security and Information Assurance, SOAR, May 8, 2009.

[IEEE 2000]

IEEE. “IEEE 100 The Authoritative Dictionary of IEEE Standards Terms Seventh Edition.” *IEEE Std 100-2000*. 2000. doi: 10.1109/IEEESTD.2000.322230.
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4116787&isnumber=4116786

[IEEE 2012]

IEEE. *About RS: IEEE Reliability Society - Reliability Engineering* (April 18, 2012).
http://rs.ieee.org/about-rs.html

[Jones 2005]

Jones, Capers. “Software Quality in 2005, a Survey of the State of the Art.” *Software Productivity Research*. Marlborough, Massachusetts, 2005.

[Krigsman 2009]
Krigsman, Michael. “Six Types of IT Project Failure.” *TechRepublic* (September 29, 2009). http://www.techrepublic.com/blog/tech-decision-maker/six-types-of-it-project-failure/

[Merriam-Webster 2013]
Merriam-Webster. *Metric* (2013). http://www.merriam-webster.com/dictionary/metric

[MITRE 2013a]
MITRE Corporation. *Common Weakness Enumeration* (2013). http://cwe.mitre.org/

[MITRE 2013b]
MITRE Corporation. *Common Attack Pattern Enumeration and Classification* (2013). http://capec.mitre.org/

[NASA 2004]
National Aero Space Administration. *NASA-STD 8739.8—Standard for Software Assurance* (2004). http://www.hq.nasa.gov/office/codeq/doctree/87398.htm

[NDIA 2011]
National Defense Industrial Association Systems Engineering Division. Working Group Report: System Development Performance Measurement—Practical Software and Systems Measurement (October 2011). http://www.psmsc.com/Downloads/Other/NDIA%20System%20Develpopment%20Performance%20Measurement%20Report.pdf

[NDIA 1999]
National Defense Industries Association Test and Evaluation Division. *Test and Evaluation Public-Private Partnership Study*. February 1999.

[NICCS 2013]
National Initiative for Cybersecurity Careers and Studies. *Interactive National Cybersecurity Workforce Framework* (2013). http://niccs.us-cert.gov/training/tc/framework

[NIST 2013]
National Institute of Standards and Technology. *Software Assurance Metrics and Tool Evaluation (SAMATE)* (2013). http://samate.nist.gov/Main_Page.html

[OMG 2013]
Object Management Group. *How to Deliver Resilient, Secure, Efficient, and Easily Changed IT Systems in Line with CISQ Recommendations* (2013). http://www.omg.org/CISQ_compliant_IT_Systemsv.4-3.pdf

[Paulk 1993]
Paulk, M.; Curtis, B.; Chrissis, M; & Weber, C. *Capability Maturity Model, Version 1.1*. Pittsburgh, PA: Carnegie Mellon University Software Engineering Institute, 1993.

[PITAC 2005]
President's Information Technology Advisory Committee. *Cybersecurity: A Crisis of Prioritization* (2005).
http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf

[SEI 2013]
Software Engineering Institute. *CERT Vulnerability Notes Dataase* (2013).
http://www.kb.cert.org/vuls

[SQA 2013]
SQA.net. *Software Quality Assurance and Software Quality Control—Terms and Definitions* (2013). http://www.sqa.net/

[Standish 2009]
Standish Group. "New Standish Group Report Shows More Project Failing and Less Successful Projects." *CHAOS Report*. Boston: Standish Group, April 23, 2009.

[Veracode 2012]
Veracode. "Study of Software Related Cybersecurity Risks in Public Companies." State of Software Security Report. April 2012.

[Whittaker 1996]
Whittaker, Brenda. *What Went Wrong? Unsuccessful Information Technology Projects.* Toronto: KPMG Consulting, 1996.

[Wikipedia 2013]
Wikipedia. *Software Metric* (2013). http://en.wikipedia.org/wiki/Software_metric

[Woodley 2013]
Woodley, M. *CS242: Programming Studio Spring 2013*.
University of Illinois, 2013. https://wiki.engr.illinois.edu/display/cs242sp13/Software+Metrics

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. **AGENCY USE ONLY**
(Leave Blank)

2. **REPORT DATE**
November 2013

3. **REPORT TYPE AND DATES COVERED**
Final

4. **TITLE AND SUBTITLE**
Software Assurance Measurement—State of the Practice

5. **FUNDING NUMBERS**
FA8721-05-C-0003

6. **AUTHOR(S)**
Dan Shoemaker & Nancy R. Mead

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**
CMU/SEI-2013-TN-019

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
AFLCMC/PZE/Hanscom
Enterprise Acquisition Division
20 Schilling Circle
Building 1305
Hanscom AFB, MA 01731-2116

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
n/a

11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

This report identifies and describes the current state of the practice in software assurance measurement. This discussion focuses on the methods and technologies that are applicable in the domain of existing software products, software services, and software processes. This report is not meant to be prescriptive; instead it attempts to provide an end-to-end discussion of the state of the practice in software assurance measurement. In addition, it points out significant emerging trends in the field. The overall discussion touches on the existing principles, concepts, methods, tools, techniques, and best practices for detection of defects and vulnerabilities in code.

14. **SUBJECT TERMS**
software assurance, measurement

15. **NUMBER OF PAGES**
54

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT**
Unclassified

18. **SECURITY CLASSIFICATION OF THIS PAGE**
Unclassified

19. **SECURITY CLASSIFICATION OF ABSTRACT**
Unclassified

20. **LIMITATION OF ABSTRACT**
UL